Using Semantic Web Technology for Self-Management of Distributed Object-Oriented Systems

Haydarlou, A.R.; Oey, M.A.; Overeinder, B.J.; Brazier, F.M.

published in
Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI-06)
2006

document version
Publisher's PDF, also known as Version of record

Using Semantic Web Technology for Self-Management of Distributed Object-Oriented Systems*

A.R. Haydarlou, M.A. Oey, B.J. Overeinder, and F.M.T. Brazier

Vrije Universiteit Amsterdam, De Boelelaan 1081a,
1081 HV Amsterdam, The Netherlands
E-mail: {rezahay,michel,bjo,frances}@cs.vu.nl

Abstract

Automated support for management of complex distributed object-oriented systems is a challenge: self-management the goal. A self-management system needs to reason about the behaviour of the distributed entities in a system, and act when necessary. The knowledge needed is multi-leveled: different levels of concepts and rules need to be represented. This paper explores the requirements that hold for representing this knowledge in self-managed distributed object-oriented systems, and explores the potential of Semantic Web technology in this context. A model for self-management knowledge and a simplified version of a real-life use case are used to illustrate the potential.

1. Introduction

Manually determining the root cause of software runtime failures in large-scale distributed heterogeneous environments is all too often current practice. Automated support would be very welcome. *Autonomic computing* [6] has been proposed as a way to reduce the cost and complexity of such systems[1], increasing manageability.

Self-management is central to autonomic computing, encompassing various self-* aspects, including self-configuring, self-healing, and self-optimising. Self-management requires systems to *know* when and where abnormal behaviour occurs (i.e., be self-aware), *analyse* the problem situation, make healing *plans*, and suggest various *solutions* to the system administrator or heal themselves. This knowledge needs to be made explicit.

Philosophers use the term *self-knowledge* to refer to knowledge one has about one's own mental states. In this paper, the term *self-management knowledge* is used to refer to (1) knowledge a system has of its internal structure and its dynamic behaviour (object-level concepts), (2) knowledge rules to reason about these object-level concepts, and (3) meta-knowledge to reason about these rules and concepts. Based on the principles of knowledge representation presented in [2], self-management knowledge representation is in fact a set of ontological commitments.

Ontologies are increasingly used to represent knowledge. For example, Stojanovic et al. [7] use ontologies to describe resources and changes in the state of a resource in a correlation engine, and Jannach et al. [5] use ontologies to describe multimedia resources and the transformation actions in a multimedia adaptation engine.

This paper explores the potential of ontological commitments (using Semantic Web technology) for self-management of distributed object-oriented systems, based on a self-management framework and a real-life case. Section 2 identifies the requirements that distributed object-oriented systems place on self-management knowledge representation, Section 3 presents an overview of Semantic Web technology, and Section 4 explores the potential of Semantic Web technology in relation to these requirements. Sections 5 through 7 explore whether self-management knowledge can be represented using Semantic Web languages. Section 8 reviews the results.

2. Requirements for the Representation of Self-Management Knowledge

This section discusses a number of important requirements for representing self-management knowledge in distributed object-oriented environments.

Knowledge locality and modularity - In a distributed system, consisting of multiple software components, the self-management knowledge of each component needs to be described and stored locally to facilitate the specification, distribution, migration, and reuse of these components.

*This research is supported by the NLnet Foundation, http://www.nlnet.nl, and Fortis Bank Netherlands, http://www.fortisbank.nl.

[1]The terms *system* and *application* will be used interchangeably.

Knowledge locality implies that a self-management system needs to be able to reason with distributed knowledge, which also puts a requirement on the representation of this self-management knowledge.

Knowledge reasoning - Self-management knowledge does not only contain concepts, but also includes rules to reason about these concepts, and even meta-rules to reason about rules themselves. Rules are used in a running self-management system to analyse a system's current status in order to detect and repair any unwanted situation. Therefore, there is a need for a rule language that supports multi-level reasoning with distributed knowledge.

Knowledge acquisition - As mentioned before, in distributed object-oriented environments knowledge is described and stored locally. This knowledge is specified by many types of users across organisations, such as system administrators, functional analysts, and system developers. To support the creation and maintenance of self-management knowledge, two non-functional requirements hold. (1) **Tool availability**: in distributed environments, knowledge acquisition tools are essential for system developers to enter, visualise, and check the consistency and soundness of complex self-management concepts, and to execute logical rules to infer new facts. (2) **User acceptance**: given the diversity in the types of users and organisations, the language used for self-management knowledge specifications should be intuitive and preferably comply with standards.

3. Semantic Web Technology Overview

The central idea of the Semantic Web initiative [1] is to augment the current web with formalised knowledge to make information on the web machine-processable. The *Semantic Web* relies on two basic technologies: (1) *ontologies* by which the domain concepts, concept hierarchies, and concept relationships can be expressed, and (2) *logical reasoning* by which new conclusions can be drawn after combining data with ontologies.

Semantic Web ontologies are expressed in a description logics language called *Ontology Web Language (OWL)*. Logical reasoning is expressed in a rule language called *Semantic Web Rule Language (SWRL)* (a W3C proposal).

In the Semantic Web hierarchy of languages, OWL is the layer above *Resource Description Framework (RDF)*. OWL combines the expressive power of description logics with the simplicity and distributive nature of RDF. SWRL extends the set of OWL axioms with Horn-like rules to enrich OWL ontologies. A rule is an implication between an antecedent (body) and consequent (head). Atoms in these rules consist of OWL concepts, properties, individuals, data values, or variables.

4. Choice for Semantic Web Ontology

Section 2 described a number of requirements for the representation of self-management knowledge. This section argues that the Semantic Web languages OWL and SWRL satisfy these requirements.

Knowledge locality and modularity - This requirement is satisfied because the Semantic Web languages support the use of *Uniform Resource Identifiers (URI)*. URIs make it possible to identify and refer to resources stored on different locations. The local knowledge, stored for each software component in a self-managed distributed system, consists of self-management concepts and rules which describe the internal structure and the behaviour of that component. These concepts and rules are considered resources and are addressable via URIs.

Knowledge reasoning - This requirement is satisfied because SWRL is specifically designed to reason about OWL concepts. SWRL is closely integrated with OWL, and therefore, rules in SWRL can directly use OWL concepts.

Knowledge acquisition - Both non-functional requirements, mentioned in Sect. 2, are satisfied by the Semantic Web languages. (1) The **tool availability** requirement is satisfied, because the Semantic Web community provides various *Integrated Development Environments (IDEs)*, plugins to other existing IDEs, Java APIs, and tools and techniques for checking the syntax and consistency of OWL documents. (2) The **user acceptance** requirement is satisfied, because the Semantic Web languages comply with W3C standards and have common and relevant features with *Unified Modeling Language (UML)*. UML is the de facto industrial standard used by software developers (users). To establish the relationship between the relevant features of UML and OWL, the *Ontology Definition Metamodel (ODM)* [3] has been defined.

In conclusion, the Semantic Web languages seem suitable to represent self-management knowledge. The next sections explore the use of these languages to express the knowledge needed in a self-management framework.

5. Self-Management Framework Overview

Figure 1 presents a high-level architecture of a self-management framework [4]. On the highest level two modules are distinguished: a managed-system and an autonomic-manager. The *managed-system* can be any existing distributed object-oriented application that has been extended with sensors and effectors. The *autonomic-manager* has two modules: (1) a self-diagnosis module, and (2) a self-adaptation module. The *self-diagnosis module* continuously checks whether the running application shows any

abnormal behaviour by monitoring the values it receives from the sensors placed in the application. If so, the self-diagnosis module determines a diagnosis and passes it to the *self-adaptation module*. This latter module is responsible for planning actions that must be taken to resolve the abnormal behaviour and uses the *effectors* to do so.

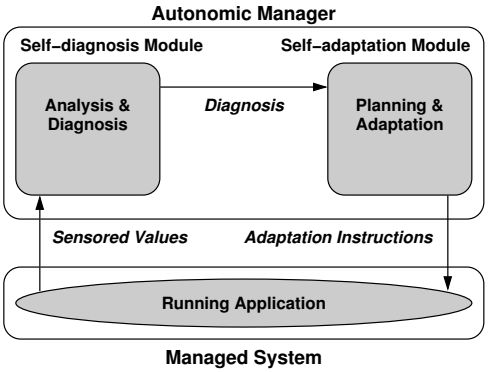

Figure 1. Self-management architecture

The framework is based on the concept of *system use-cases* which describe the response of a system to a given request [4]. To monitor a system use-case realisation and to repair abnormal behaviour, the framework needs knowledge regarding the *internal structure* and *dynamic behaviour* of the running application.

6. Self-Management Concepts

Figure 2 illustrates the high-level self-management concepts of the framework, related to each other by means of OWL object properties. For the sake of space and simplicity, only relevant OWL object properties (without OWL property restrictions) are depicted.

The AutonomicManager monitors the execution of a system use-case. It has a containment relation with the concepts Analyser, Diagnoser, Planner, and PlanExecutor. The Analyser is responsible for detecting an incorrect system use-case realisation (i.e., abnormal system behaviour). A Symptom represents such abnormal system behaviour based on one or more sensor-values. A Sensor is instrumented in a software unit (ManagedElement) in a running application.

The Diagnoser uses the determined Symptom and its own meta-knowledge to produce a Diagnosis addressing the possible root-cause of application malfunctioning. The Planner constructs a Plan based on the Diagnosis. A Plan is an ordered set of actions (self-configuring, self-healing, or self-optimising actions) needed to repair the detected abnormal behaviour. PlanExecutor is responsible for translating Plans into Effectors which are instrumented in ManagedElements to adapt the system's behaviour by executing the plan.

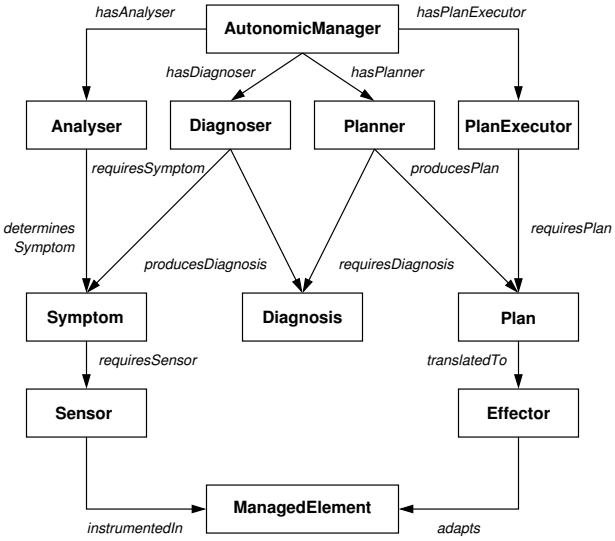

Figure 2. Self-management overview

7. Self-Management Sub-Ontologies

The self-management ontology is modular and extensible. It is composed of a number of sub-ontologies each of which contains their own reusable knowledge. The sub-ontologies are: autonomic-manager, analyser, diagnoser, planner, plan-executor, dynamic-model, sensor, effector, and static-model. A number of these sub-ontologies are briefly described and illustrated in the context of a distributed trading application scenario, borrowed from Fortis Bank Netherlands.

This paper presents a much simplified version of the trading application, consisting of only two subsystems: a web-application and a banking-service that communicate with each other with the *SOAP* protocol. The web-application captures the trade information (such as trade-date, trade-amount, customer-account, fund-account), entered in a web page by users, and sends it to the banking-service that performs the actual money transfer between different bank accounts.

7.1. Static-Model Sub-Ontology

The *static-model sub-ontology* provides a description of the internal structure of an application. Distributed object-oriented applications are usually composed of one or more subsystems, each of which is composed of a number of components. Components either contain other (sub)components or a number of classes. The ManagedElement concept represents each of these application parts. Each ManagedElement has a ManagedElementState, and a list of Connectors that bind one ManagedElement to another. A State models a data item whose value may change

during application lifetime and that is important enough to monitor. There are four different types of managed-elements: ManagedSystem, ManagedRunnable, ManagedComponent, and ManagedClass. A ManagedSystem is used to describe and manage the collective behaviour of a number of related subsystems. It is composed of a number of ManagedRunnables. A ManagedRunnable models a part of an application that is runnable (such as a subsystem or an execution thread), that can be started/stopped. A ManagedRunnable is composed of one or more ManagedRunnables or ManagedComponents, each of which models a software component or library (such as a logging component). A ManagedComponent is composed of one or more ManagedComponents or ManagedClasses. ManagedClass is an atomic managed-element that corresponds with a coding-level class and models the static properties of that class (such as class name, class file, and file location).

Figure 3 shows an example of the internal structure of the simplified trading application, simplifiedTrading, which consists of two ManagedRunnables: webApp and bankingApp. The two ManagedRunnables are connected to each other by the webToBankingConn connector which uses the *SOAP* protocol (soapProtocol). The connector has a state (connAvailability) that can be monitored to see whether the connection is available. The webApp ManagedRunnable contains two ManagedComponents: paymentComp and webInterfaceComp. The paymentComp ManagedComponent, in turn, contains two ManagedClasses: tradeCls and customerCls. The tradeCls ManagedClass has a state (amount) which can be monitored to see whether its value exceeds a given threshold.

```
<ManagedSystem rdf:ID="simplifiedTrading">
    <hasSubElement rdf:resource="#webApp"/>
    <hasSubElement rdf:resource="#bankingApp"/>
</ManagedSystem>
<AtomicManagedRunnable rdf:ID="webApp">
    <hasSubElement rdf:resource="#paymentComp"/>
    <hasSubElement rdf:resource="#webInterfaceComp"/>
    <hasConnector rdf:resource="#webToBankingConn"/>
</AtomicManagedRunnable>
<RunnableToRunnableConnector rdf:ID="webToBankingConn">
    <hasProtocol rdf:resource="#soapProtocol"/>
    <hasConnectorState
        rdf:resource="dynamic-model#connAvailability"/>
    <hasFirstElement rdf:resource="#webApp"/>
    <hasSecondElement rdf:resource="#bankingApp"/>
</RunnableToRunnableConnector>
<SOAPProtocol rdf:ID="soapProtocol"/>
<AtomicManagedComponent rdf:ID="paymentComp">
    <hasSubElement rdf:resource="#tradeCls"/>
    <hasSubElement rdf:resource="#customerCls"/>
</AtomicManagedComponent>
<ManagedClass rdf:ID="tradeCls">
    <hasManagedElementState
        rdf:resource="dynamic-model#amount"/>
</ManagedClass>
```

Figure 3. Static-model ontology example

7.2. Dynamic-Model Sub-Ontology

The *dynamic-model sub-ontology* offers a description of the dynamic behaviour (system use-case realisation) of an application, and consists of the specification of a collection of Jobs, Tasks, Symptoms, States, and Events. A system use-case is represented as a Job. Each Job contains a number of various Tasks. Tasks are the basic units of a dynamic application model. Conceptually, both jobs and tasks read input, manipulate data, and produce output. StateSensors and EventSensors associated with a Task monitor state changes or event occurrences within the execution of that task.

There are three types of Jobs: OperationalJob, FunctionalJob, and ImplementationalJob. An OperationalJob describes a system use-case realisation from the viewpoint of a *system administrator*. According to this view, a system is composed of a number of processes and threads (ManagedRunnables) that cooperate with each other to realise a system use-case. A FunctionalJob describes a system use-case realisation from the viewpoint of a *functional analyst*. According to this view, a system is composed of a number of functional components (ManagedComponents) each of which is responsible for realising a specific part of some business functionality. An ImplementationalJob describes a system use-case realisation from the viewpoint of a *system developer*. According to this view, a system is composed of the low level code bundled in classes (ManagedClasses).

Figure 4 shows the specification of an example ImplementationalJob, sendPayment, that contains one Task, amountManipulation. The possible incorrect behaviour of this job is described by one Symptom, lowAmountSymptom. The amountManipulation task reads the state (amount) defined in the ManagedClass tradeCls as a variable of type *double*. The execution of this task is monitored by the amountSensor defined in the sensor sub-ontology.

```
<ImplementationalJob rdf:ID="sendPayment">
    <hasTask rdf:resource="#amountManipulation"/>
    <hasSymptom rdf:resource="analyser#lowAmountSymptom"/>
 </ImplementationalJob>
<ManagedClassDynamicStateManipulation
        rdf:ID="amountManipulation">
    <manipulatedState rdf:resource="#amount"/>
    <executesWithin rdf:resource="static-model#tradeCls"/>
    <monitoredBy rdf:resource="sensor#amountSensor"/>
</ManagedClassDynamicStateManipulation>
<ManagedClassDynamicState rdf:ID="amount">
    <hasStateType rdf:resource="#doubleType"/>
    <hasStateOperation rdf:resource="#readOperation"/>
</ManagedClassDynamicState>
<ConnectorState rdf:ID="connAvailability">
    <hasStateType rdf:resource="#enumType"/>
    <hasStateOperation rdf:resource="#readOperation"/>
</ConnectorState>
```

Figure 4. Dynamic-model ontology example

7.3. Sensor Sub-Ontology

The *sensor sub-ontology* describes various Sensor types which retrieve runtime information from the running application. On the highest level, there are two types of Sensors: StateSensors and EventSensors. A StateSensor is used to monitor any data item that is read or written by a Task. An EventSensor monitors the occurrence of an Event during execution of a Task.

```
<ManagedClassDynamicStateSensor rdf:ID="amountSensor">
    <monitorsItem rdf:resource="dynamic-model#amount"/>
    <triggeredByTask
        rdf:resource="dynamic-model#amountManipulation"/>
    <observedValue rdf:datatype="xsd#double">0</observedValue>
</ManagedClassDynamicStateSensor>
```

Figure 5. Sensor ontology example

Figure 5 shows the specification of an example Sensor (amountSensor), which describes the monitoring of the variable (amount) during execution of the amountManipulation task. In runtime, the actual observedValue of the variable amount (whose default value has been set to zero) is passed to the associated software module that manages the Symptom concept.

7.4. Analyser Sub-Ontology

The *analyser sub-ontology* describes how an Analyser analyses a Job execution, and how incorrect behaviour of a Job can be expressed as Symptoms. There are three types of Symptoms, corresponding with the three Job types, mentioned before: OperationalSymptom, FunctionalSymptom, and ImplementationalSymptom.

Figure 6 shows the specification of an example Analyser (tradeAnalyser), which analyses the execution of the ImplementationalJob sendPayment and determines the occurrence of the ImplementationalSymptom lowAmountSymptom. The lowAmountSymptom has a boolean property (hasArisen) which indicates whether the symptom has occurred. The value of the boolean property hasArisen is determined by the SWRL rule lowAmountSymptomRule. The Human Readable Syntax of the lowAmountSymptomRule rule is as follows:

```
observedValue(amountSensor, ?x) ∧
                    swrlb:lessThan(?x, 20.0)
  → hasArisen(lowAmountSymptom, true)
```

The rule states that if the value of the amount state, retrieved from the amountSensor, is less than 20.0 then the lowAmountSymptom has arisen.

```
<Analyser rdf:ID="tradeAnalyser">
    <analysesJob rdf:resource="dynamic-model#sendPayment"/>
    <determinesSymptom rdf:resource="#lowAmountSymptom"/>
</Analyser>
<ImplementationalSymptom rdf:ID="lowAmountSymptom">
    <requiresSensor rdf:resource="sensor#amountSensor"/>
    <swrl:Imp rdf:ID="lowAmountSymptomRule">
    ...
    </swrl:Imp>
</ImplementationalSymptom>
```

Figure 6. Analyser ontology example

8. Review of Results

Research communities from different disciplines are showing increasing interest in Semantic Web technology for knowledge representation. This paper explores the potential of this technology for self-management of distributed object-oriented systems: for a specific self-management framework and a specific real-life case.

OWL and SWRL are succesfully used to express the ontological commitments needed to represent self-management knowledge. OWL is used to express knowledge about self-management concepts and concept hierarchies. SWRL is used to express system behaviour and multi-level diagnostic reasoning. Their ability to represent knowledge relates strongly to the requirements of representing self-management knowledge in distributed object-oriented systems: support for knowledge acquisition, local knowledge representation, and distributed reasoning.

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[2] R. Davis, H. E. Shrobe, and P. Szolovits. What is a knowledge representation? *AI Magazine*, 14(1):17–33, 1993.

[3] L. Hart, P. Emery, R. M. Colomb, K. Raymond, D. Chang, Y. Ye, E. Kendall, and M. Dutra. Usage scenarios and goals for ontology definition metamodel. In *WISE*. Springer-Verlag, 2004.

[4] A. R. Haydarlou, B. J. Overeinder, M. A. Oey, and F. M. T. Brazier. Multi-level model-based self-diagnosis of distributed object-oriented systems. In *Proceedings of the 3rd IFIP International Conference on Autonomic and Trusted Computing (ATC-06)*, Wuhan, China, September 2006.

[5] D. Jannach, K. Leopold, C. Timmerer, and H. Hellwagner. A knowledge-based framework for multimedia adaptation, 2006.

[6] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[7] L. Stojanovic, J. Schneider, A. Maedche, S. Libischer, R. Studer, A. Abecker, G. Breiter, and J. Dinger. The role of ontologies in autonomic computing systems. *IBM Systems Journal*, 43(3), Aug 2004.